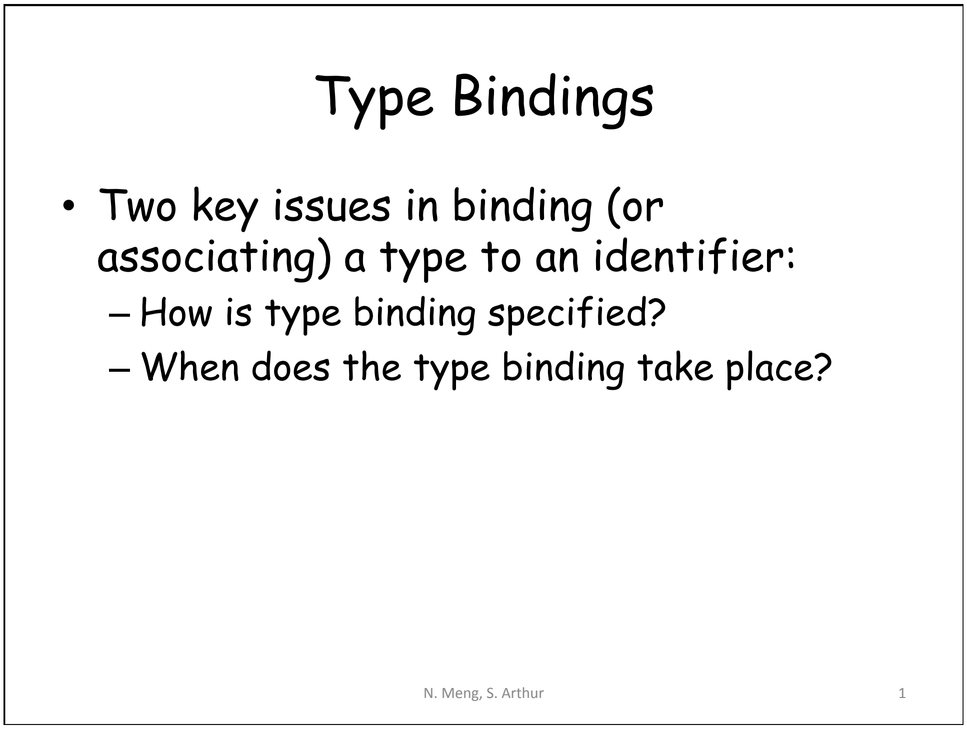
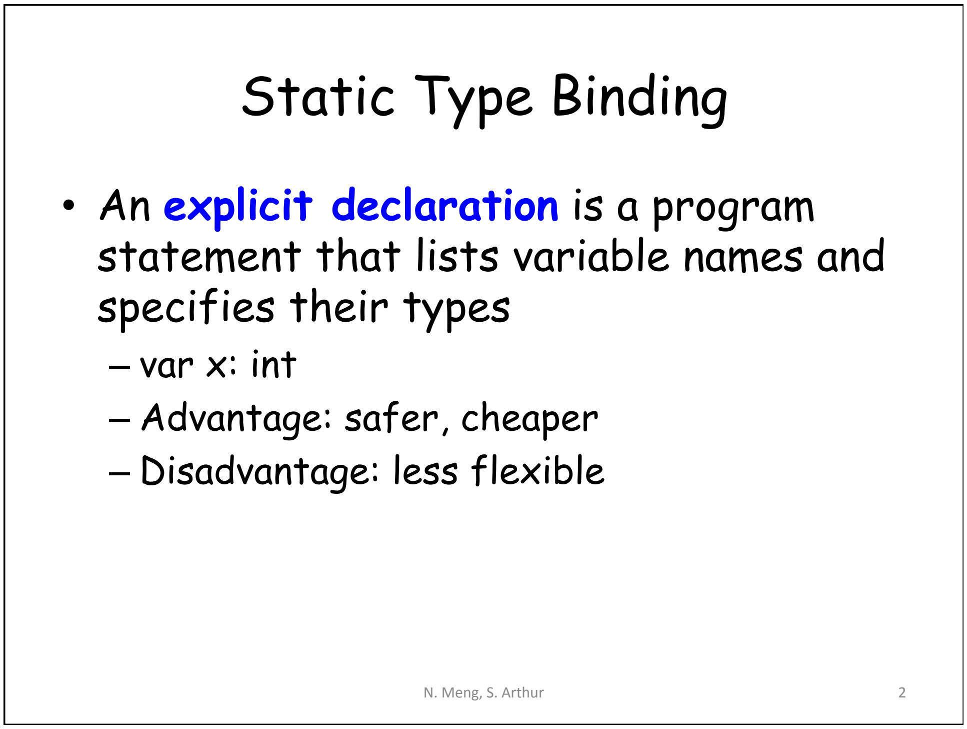


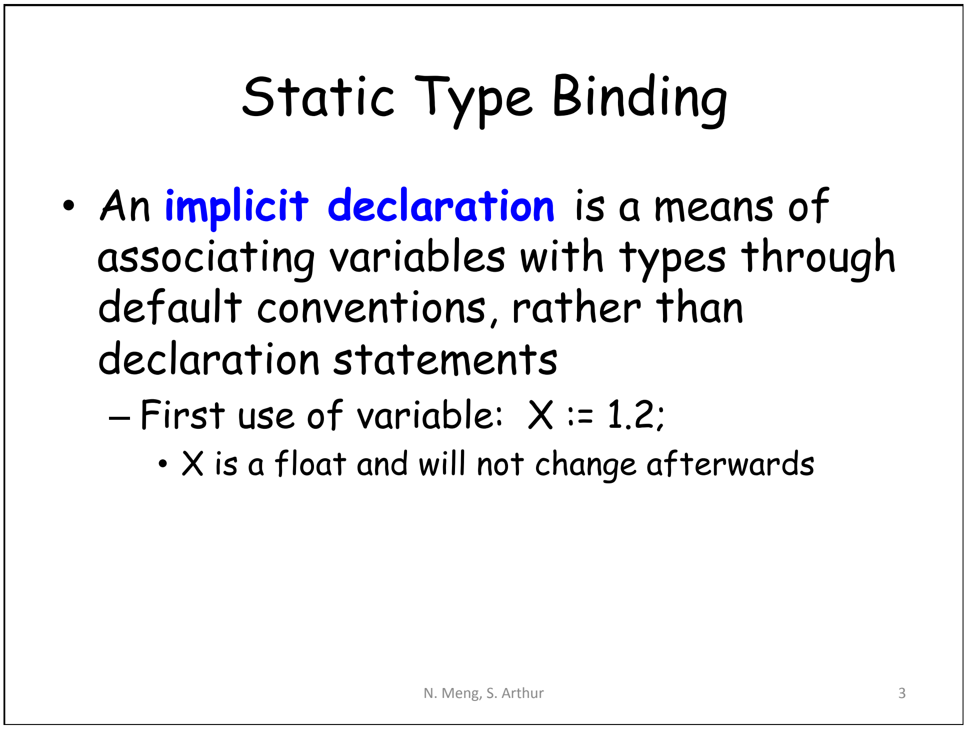
Type Bindings

- Two key issues in binding (or associating) a type to an identifier:
  - How is type binding specified?
  - When does the type binding take place?



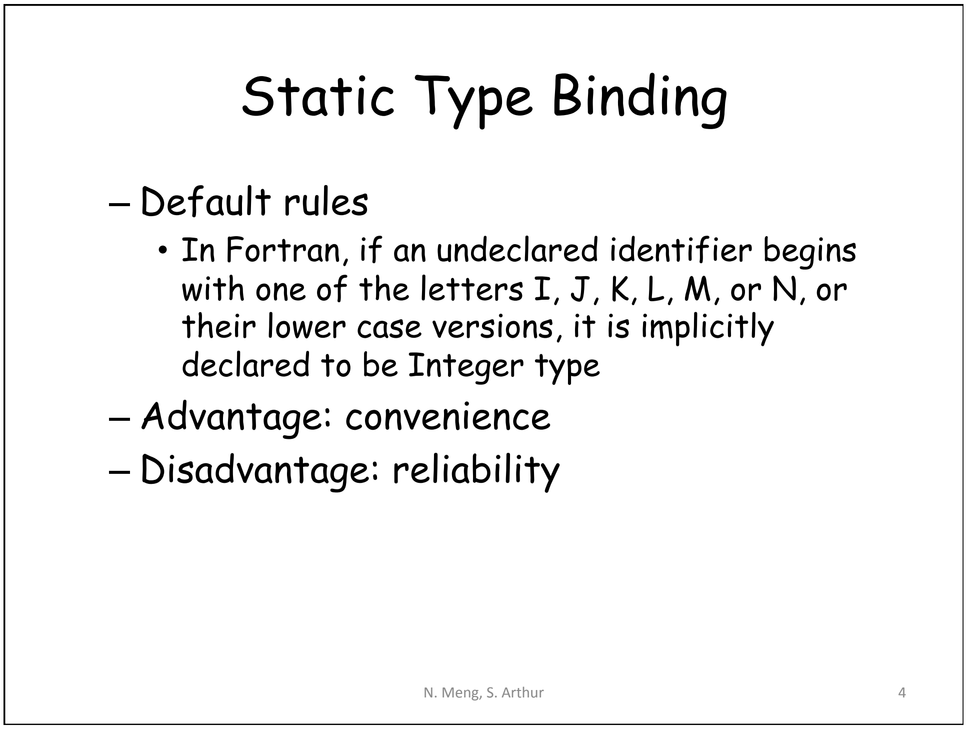
Static Type Binding

- An **explicit declaration** is a program statement that lists variable names and specifies their types
  - var x: int
  - Advantage: safer, cheaper
  - Disadvantage: less flexible



Static Type Binding

- An **implicit declaration** is a means of associating variables with types through default conventions, rather than declaration statements
  - First use of variable: X := 1.2;
    - X is a float and will not change afterwards



Static Type Binding

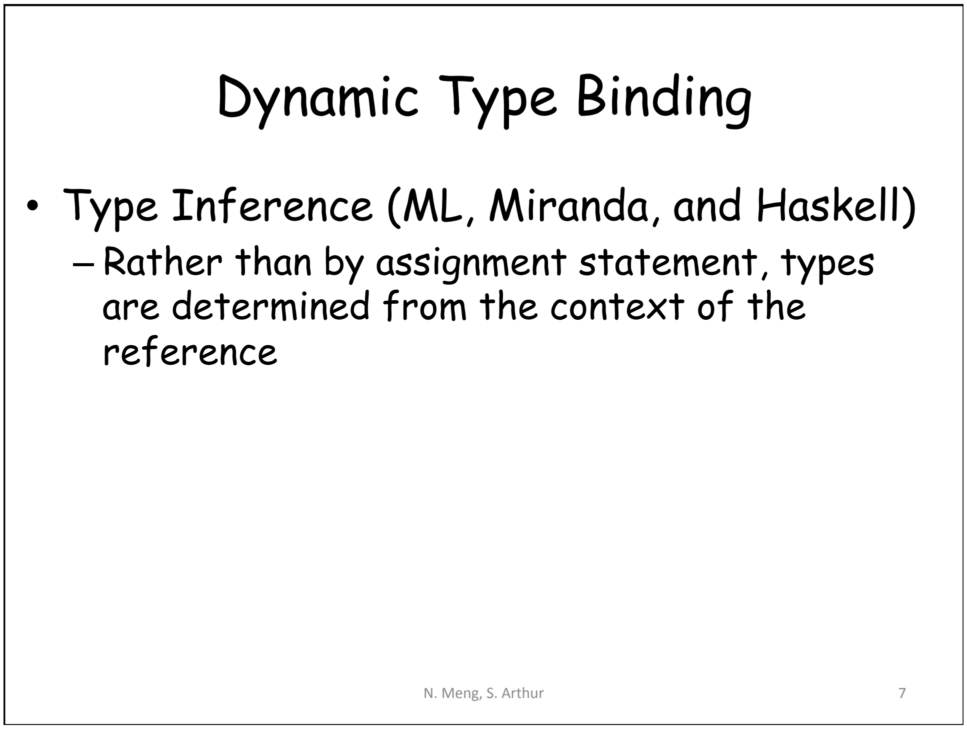
- Default rules
  - In Fortran, if an undeclared identifier begins with one of the letters I, J, K, L, M, or N, or their lower case versions, it is implicitly declared to be Integer type
- Advantage: convenience
- Disadvantage: reliability

Dynamic Type Binding

- The type of a variable is not specified by a declaration statement, nor can it be determined by the spelling of its name
- Instead, the variable is bound to a type when it is assigned a value in an assignment statement
  - E.g., list = [10. 2, 3.5]; (JavaScript)
  - Regardless of its previous type, list has the new type of single-dimension array of length 2

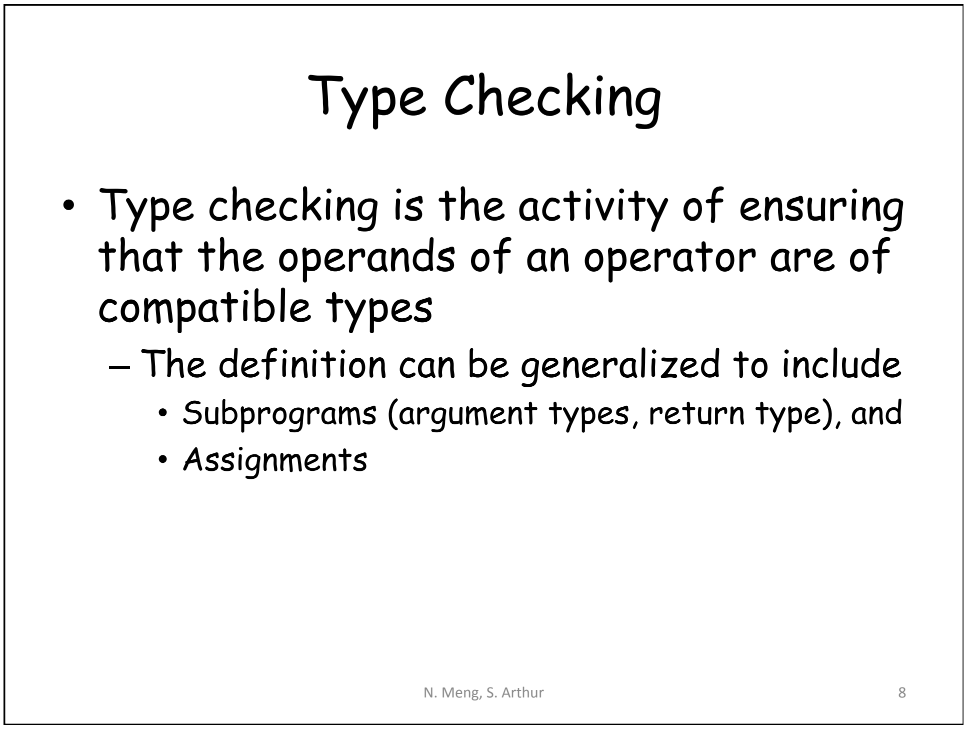
Dynamic Type Binding

- Advantage
  - flexibility (can change type dynamically)
- Disadvantage
  - Type error detection by the compiler is difficult
  - High cost
    - Type checking must be done at runtime
    - Every variable must have a runtime descriptor to maintain the current type
    - The storage used for the value of a variable must be of varying size



Dynamic Type Binding

- Type Inference (ML, Miranda, and Haskell)
  - Rather than by assignment statement, types are determined from the context of the reference



Type Checking

- Type checking is the activity of ensuring that the operands of an operator are of compatible types
  - The definition can be generalized to include
    - Subprograms (argument types, return type), and
    - Assignments

Type Checking

- A compatible type is one that
  - is legal for the operator, or
  - is allowed under language rules to be implicitly converted to a legal type
    - The automatic conversion is called (implicit) coercion
    - Mixed mode arithmetic (2 + 3.5)

Type Error

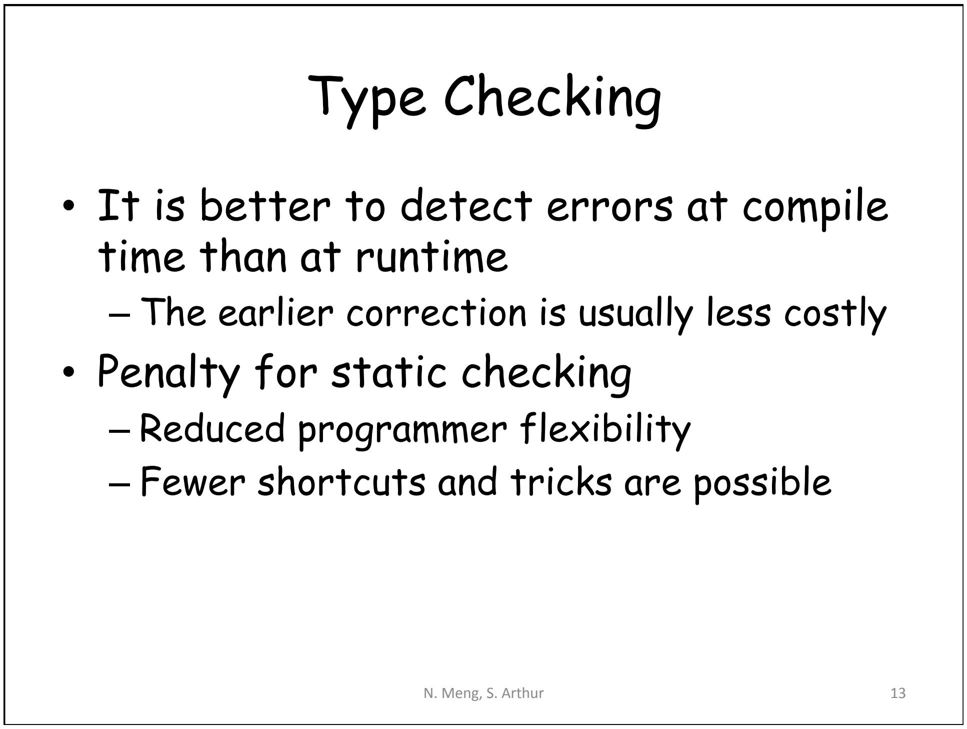
- A type error is the application of an operator to an operand of an inappropriate type
  - 1.5 + “Just say NO! to UVA”

Type Checking

- If all bindings of variables to types are static in a language, then type checking can nearly always be done statically
- Dynamic type binding requires type checking at runtime, which is called **dynamic type checking**
  - Dynamic type binding only allows dynamic type checking

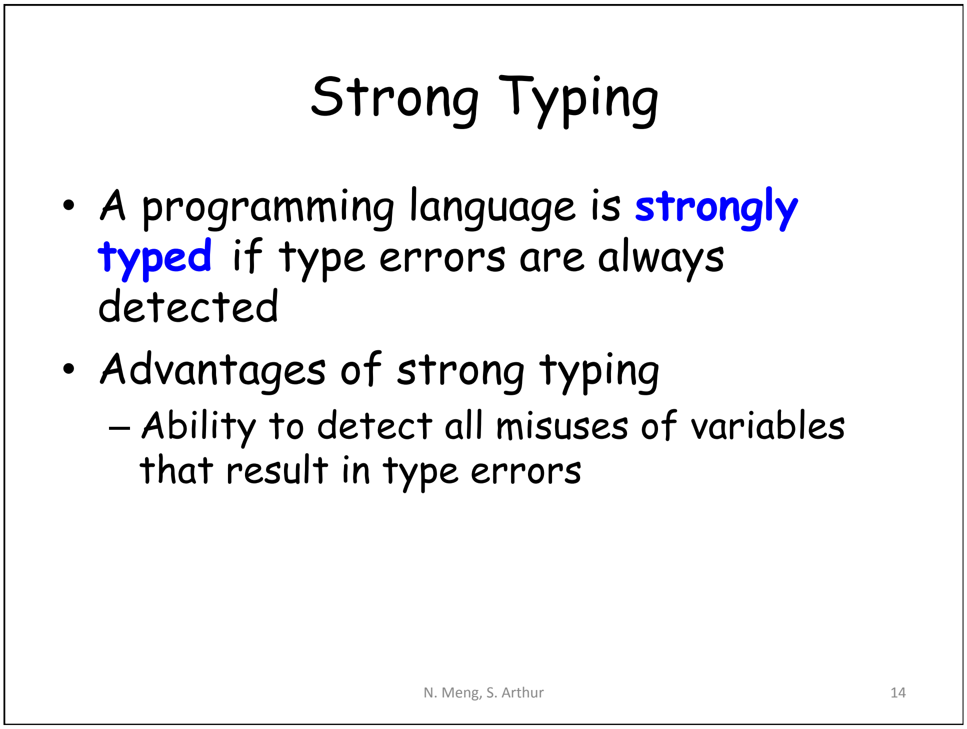
Type Checking

- Type checking is complicated when a language allows a memory cell to store values of different types at different times during execution
  - E.g., C and C++ unions
  - In such cases, type checking must be dynamic
- Even though all variables are statically bound to types, not all type errors can be detected by static type checking



Type Checking

- It is better to detect errors at compile time than at runtime
  - The earlier correction is usually less costly
- Penalty for static checking
  - Reduced programmer flexibility
  - Fewer shortcuts and tricks are possible



Strong Typing

- A programming language is **strongly typed** if type errors are always detected
- Advantages of strong typing
  - Ability to detect all misuses of variables that result in type errors

Language Comparison for Strong Typing

- FORTRAN 95 is not strongly typed
  - the use of Equivalence between variables of different types allows a variable of one type to refer to a value of a different type
- C and C++ are not strongly typed
  - Both include union types, which are not type checked

Language Comparison for Strong Typing

- Ada, Java, and C# are almost strongly typed
  - It allows programmers to breach the type-checking rules by specially requesting that type checking be suspended for a particular type conversion
- ML is strongly typed

Coercion Rules

- Coercion rules can weaken strong typing
  - E.g., int a = 3, b = 5;
    float d = 4.5;
  - If a developer meant to type a + b, but mistakenly typed a + d, the error would not be detected by the compiler due to coercion
- Languages with more coercion are less reliable than those with little coercion
  - Reliability comparison
    - Fortran/C/C++ < Ada
    - C++ < Java/C#

Type Compatibility

- The rules dictate the type of operands that are acceptable for each operator and thereby specify the possible type errors of the language
- Type rules are called compatibility because in some cases, the type of an operand can be implicitly converted by the compiler or runtime system to make it acceptable to the operator

Type Equivalence

- A strict form of type compatibility—compatibility without coercion
- Two approaches to defining type equivalence
  - Name type equivalence (Type equivalence by name)
  - Structure type equivalence (Type equivalence by structure)

Name Type Equivalence

- Two variables have equivalent types if they are defined in the same declaration or in declarations using the same type name
  - Ex. 1, int a, b;
  - Ex. 2, int a; int b;

Name Type Equivalence

- Easy to implement but is more restrictive
  - In Ada

```
type Indextype is 1..100;
count : Integer;
index: IndexType;
```

- The type of count is a subrange of the integers, which is not equivalent to the integer type
- The two variables cannot be assigned to each other

Name Type Equivalence

- In Pascal

```
Type X: array[1..5] of integer
Y: X;
Procedure K(J: array[1..5] of integer
…
K(Y)   /* Y incompatible with J */
```

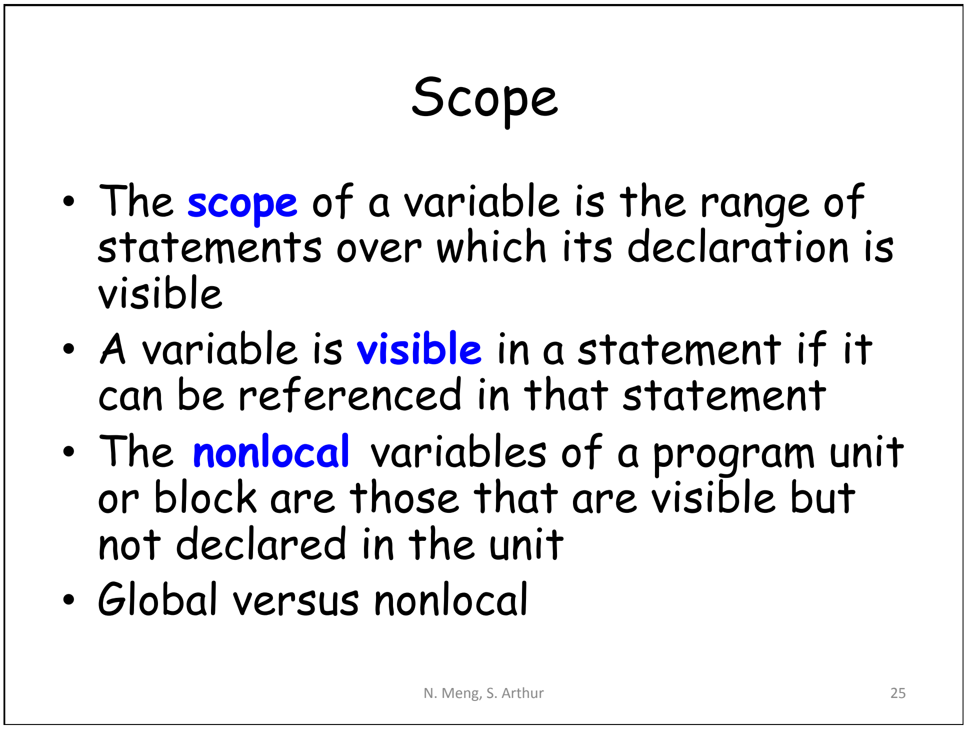
- Although J and X have the same type structure, they are considered as two types
- Y cannot be passed as a valid parameter to call K

Structure Type Equivalence

- Two variables have equivalent types if their types have identical structures
  - Ex 1., type celsius = float;
                    fahrenheit = float;
  - The two types are considered equivalent

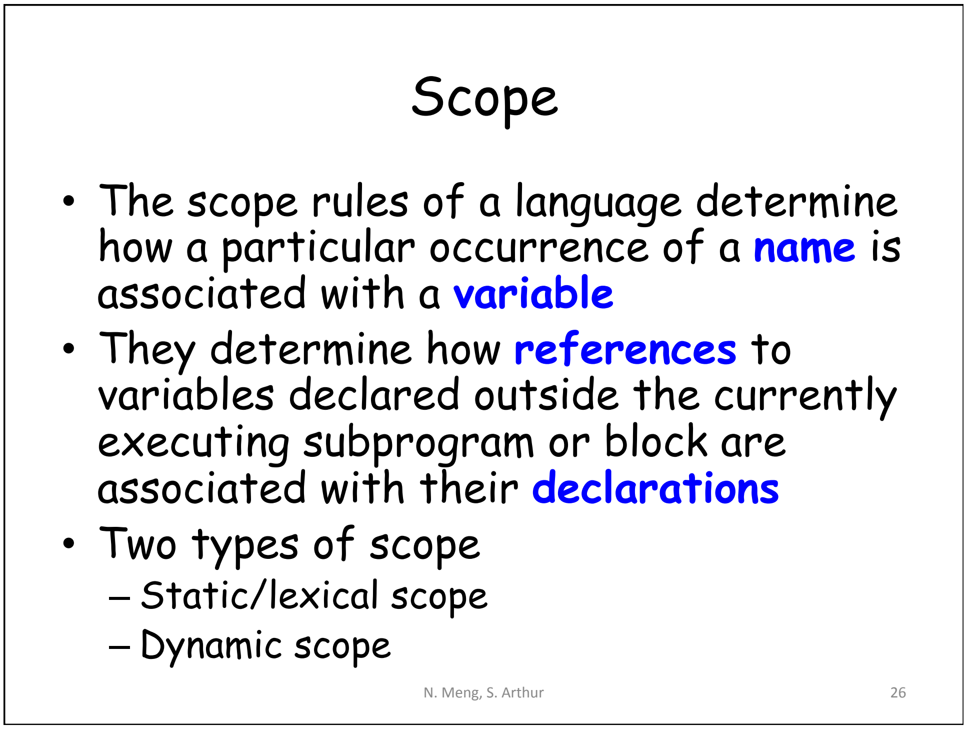
Structure Type Equivalence

- More flexible, but harder to implement
  - The entire structures of two types must be compared
- Developers are not allowed to differentiate between types with the same structure



Scope

- The **scope** of a variable is the range of statements over which its declaration is visible
- A variable is **visible** in a statement if it can be referenced in that statement
- The **nonlocal** variables of a program unit or block are those that are visible but not declared in the unit
- Global versus nonlocal



Scope

- The scope rules of a language determine how a particular occurrence of a **name** is associated with a **variable**
- They determine how **references** to variables declared outside the currently executing subprogram or block are associated with their **declarations**
- Two types of scope
  - Static/lexical scope
  - Dynamic scope

Static Scope

- The scope of a variable can be statically determined, that is, prior to execution
- Two categories of static-scoped languages
  - Languages allowing nested subprograms: Ada, JavaScript, and PHP
  - Languages which does not allow subprograms: C-based languages

Static Scope

- To connect a name reference to a variable, you must find the **appropriate declaration**
- Search process
  1. search the declaration locally
  2. If not found, search the next-larger enclosing unit
  3. Loop over step 2 until a declaration is found or an undeclared variable error is detected

Static Scope

- Given a subprogram Sub1,
  - the subprogram that declared Sub1 is called its **static parent**
  - the static parent of Sub1, its static parent, and so forth up to and including the largest enclosing subprogram, are called **static ancestors** of Sub1

An Example (Ada)

```
1. procedure Big is
2.   X : Integer;
3.   procedure Sub1 is
4.        X: Integer;
5.        begin  -- of Sub1
6.        …
7.        end;    -- of Sub1
8.   procedure Sub2 is
9.        begin   -- of Sub2
10.       … X …
11.       end;      -- of Sub2
12. begin           -- of Big
13. …
14. end;            -- of Big
```

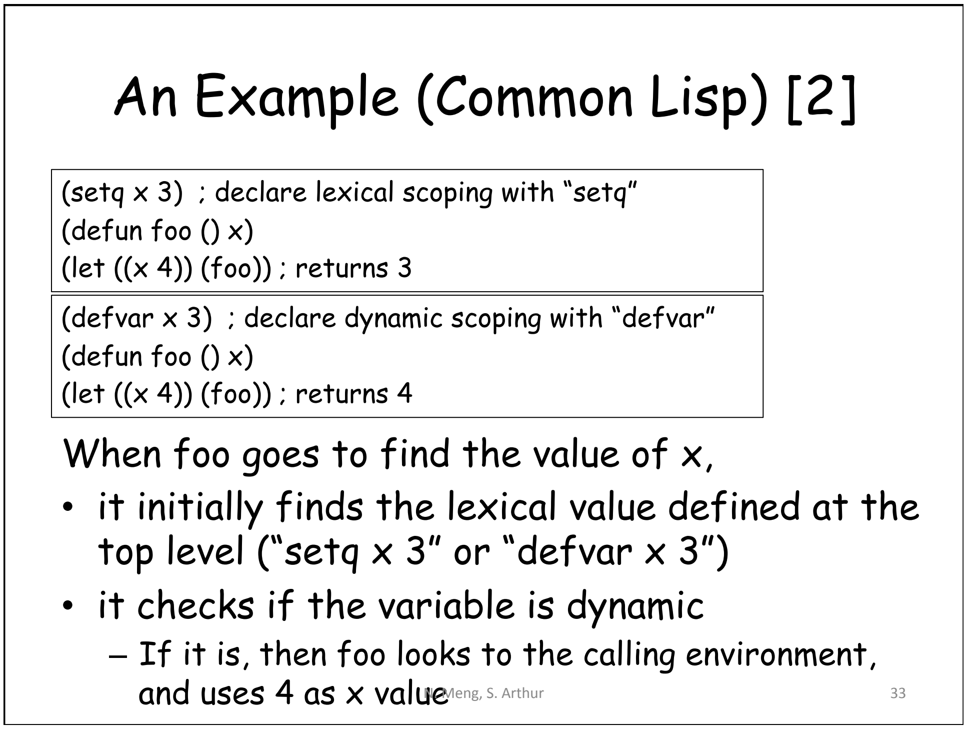
- Which declaration does X in line 10 refer to?

Variable Hiding

- Variables can be hidden from a unit by having a “closer” variable with the same name
  - “Closer” means more immediate enclosing scope
  - C++ and Ada allow access to the “hidden” variables (using fully qualified names)
    - scope.name
- Blocks can be used to create new static scopes inside subprograms

Dynamic Scope

- Dynamic scoping is based on the calling sequence of subprograms, not on their spatial relationship to each other
- Dynamic scope can be determined only at runtime
- Always used in interpreted languages, which usually does not have type checking at compile time



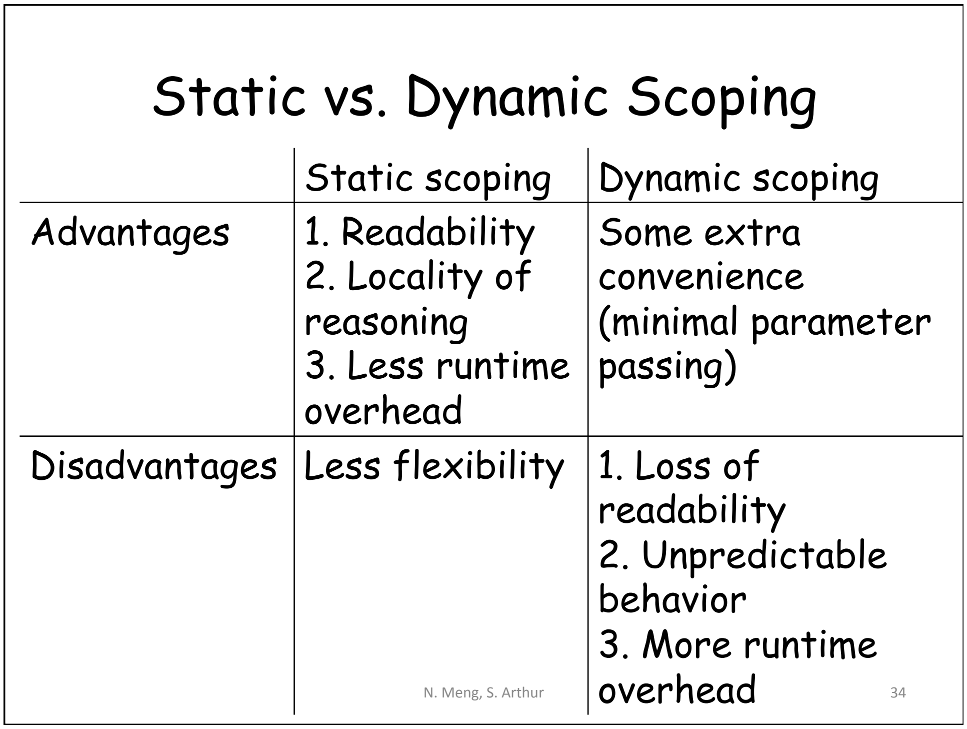
An Example (Common Lisp) [2]

```
(setq x 3)  ; declare lexical scoping with "setq"
(defun foo () x)
(let ((x 4)) (foo)) ; returns 3
```

```
(defvar x 3)  ; declare dynamic scoping with "defvar"
(defun foo () x)
(let ((x 4)) (foo)) ; returns 4
```

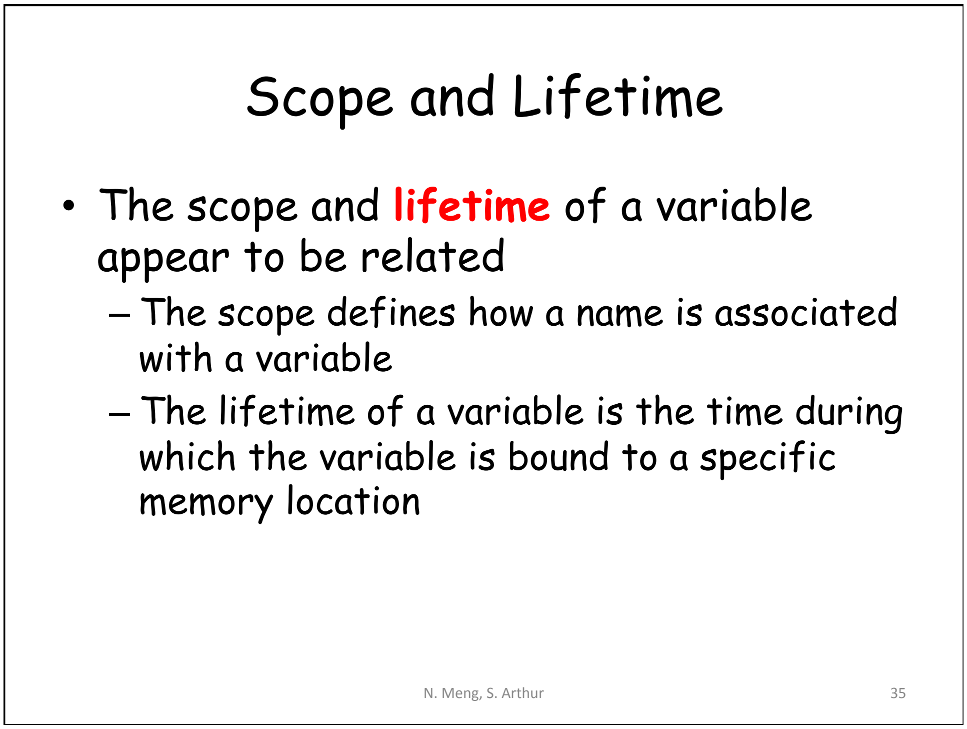
When foo goes to find the value of x,

- it initially finds the lexical value defined at the top level ("setq x 3" or "defvar x 3")
- it checks if the variable is dynamic
  - If it is, then foo looks to the calling environment, and uses 4 as x value



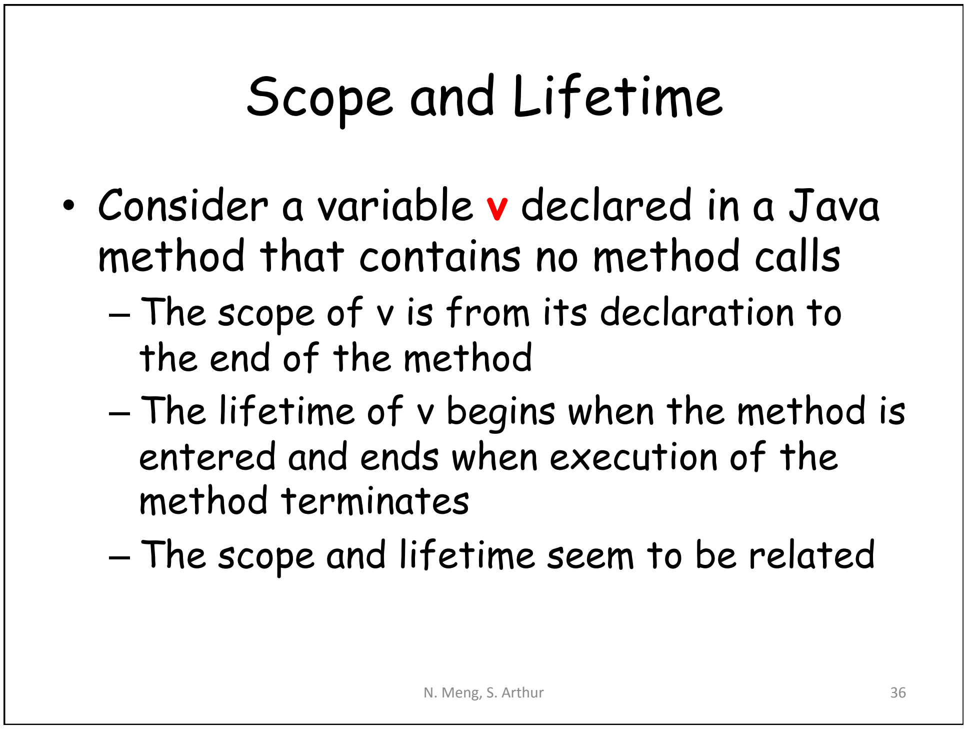
Static vs. Dynamic Scoping

	Static scoping	Dynamic scoping
Advantages	1. Readability 2. Locality of reasoning 3. Less runtime overhead	Some extra convenience (minimal parameter passing)
Disadvantages	Less flexibility	1. Loss of readability 2. Unpredictable behavior 3. More runtime overhead



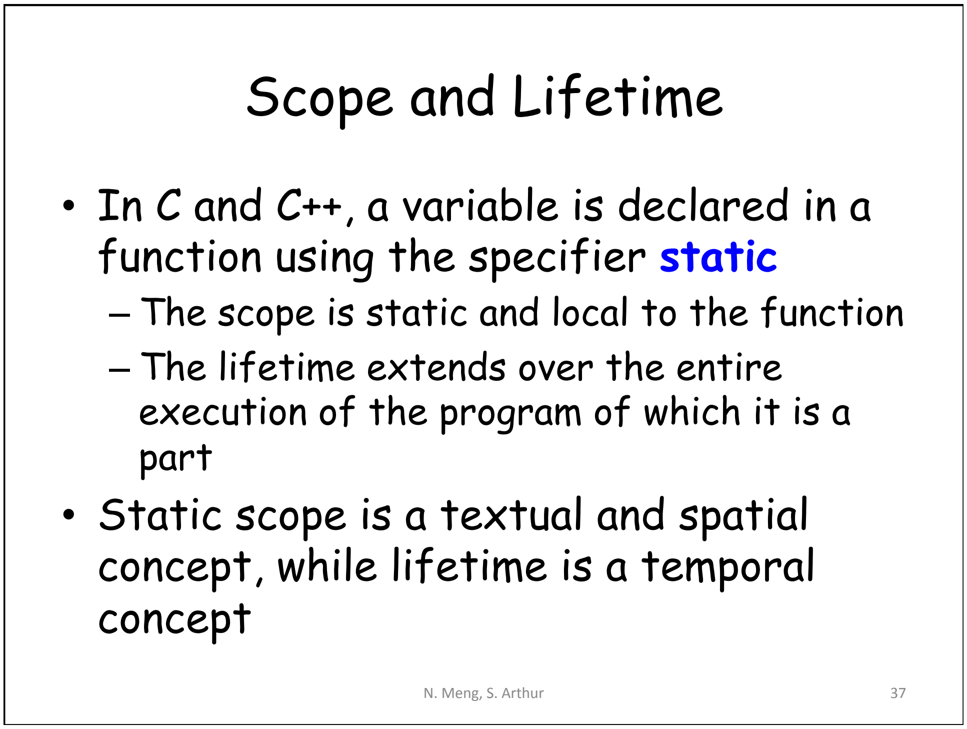
Scope and Lifetime

- The scope and **lifetime** of a variable appear to be related
  - The scope defines how a name is associated with a variable
  - The lifetime of a variable is the time during which the variable is bound to a specific memory location



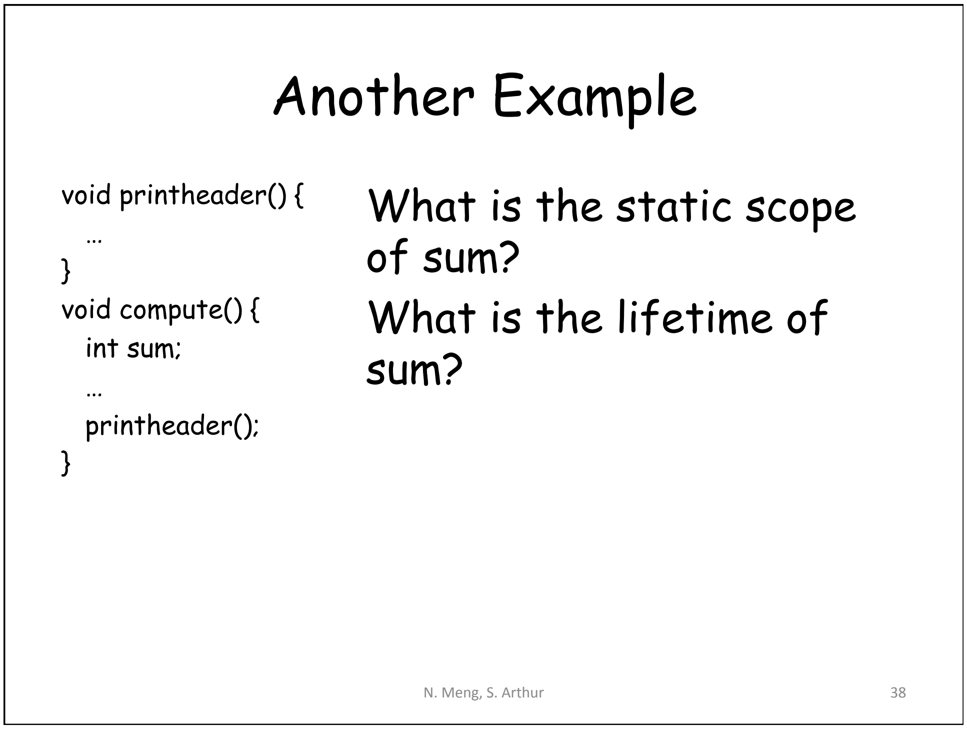
Scope and Lifetime

- Consider a variable **v** declared in a Java method that contains no method calls
  - The scope of v is from its declaration to the end of the method
  - The lifetime of v begins when the method is entered and ends when execution of the method terminates
  - The scope and lifetime seem to be related



Scope and Lifetime

- In C and C++, a variable is declared in a function using the specifier **static**
  - The scope is static and local to the function
  - The lifetime extends over the entire execution of the program of which it is a part
- Static scope is a textual and spatial concept, while lifetime is a temporal concept

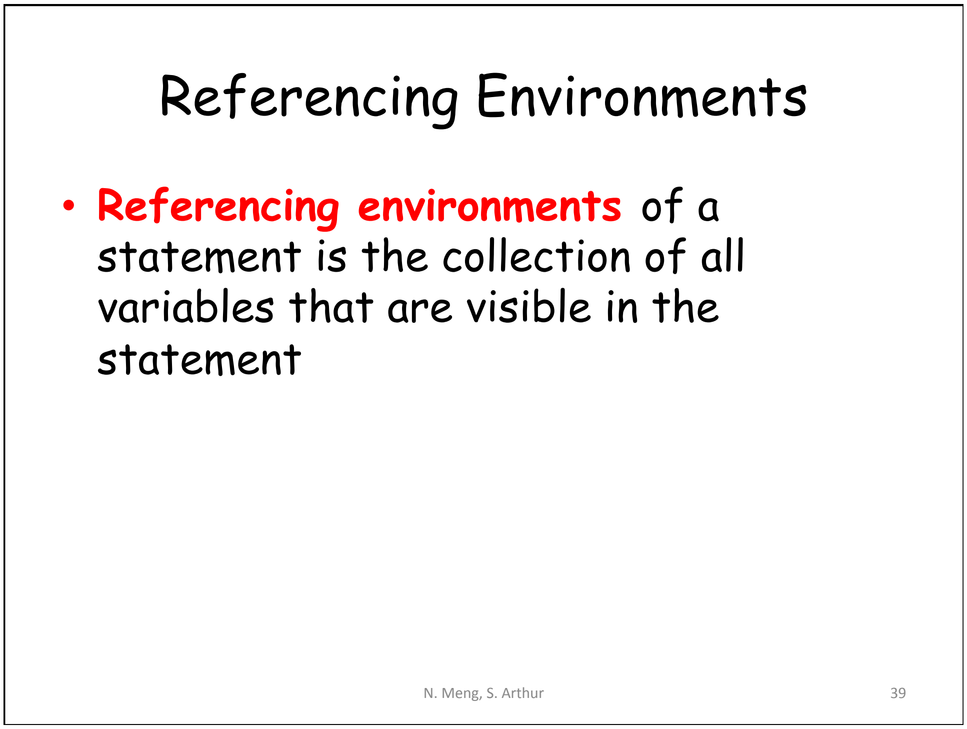


Another Example

```
void printheader() {
   …
}
void compute() {
   int sum;
   …
   printheader();
}
```

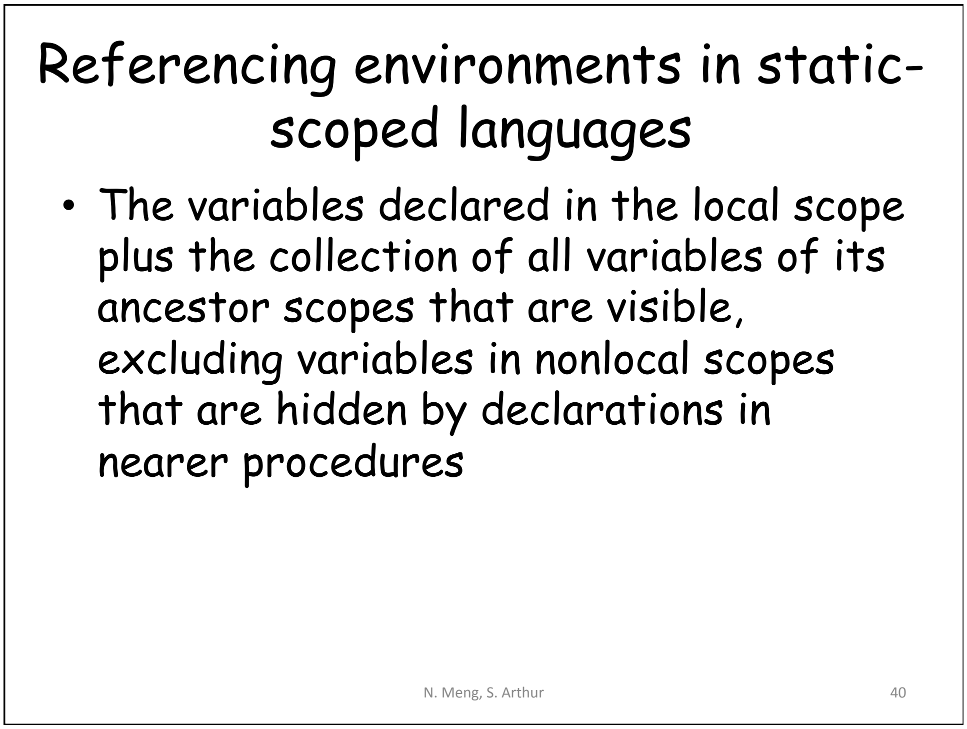
What is the static scope of sum?

What is the lifetime of sum?



Referencing Environments

- **Referencing environments** of a statement is the collection of all variables that are visible in the statement



Referencing environments in static-scoped languages

- The variables declared in the local scope plus the collection of all variables of its ancestor scopes that are visible, excluding variables in nonlocal scopes that are hidden by declarations in nearer procedures

An Example

```
procedure Example is
   A, B : Integer;
   …  ←------------------------1
   procedure Sub1 is
      X, Y: Integer;
      begin  -- of Sub1
         …     ←---------------2
      end;   -- of Sub1
   procedure Sub2 is
     X: Integer;
     begin  -- of Sub2
     …         ←---------------3
     end;   -- of Sub2
begin         -- of Example
…  ←----------------------4
end;          -- of Example
```

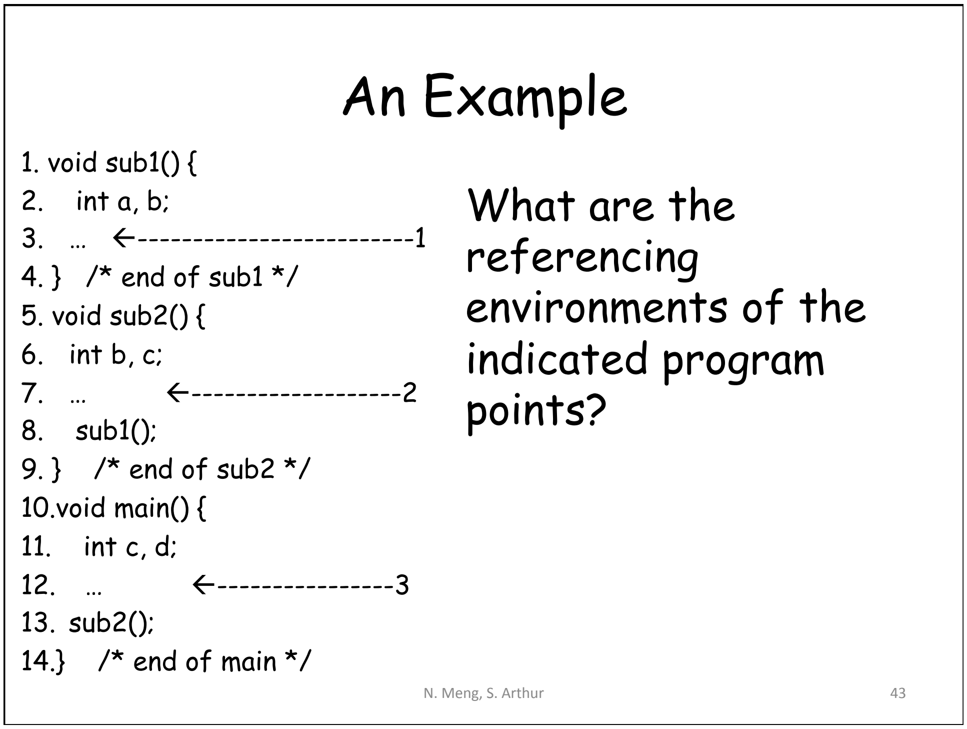
What are the referencing environments of the indicated program points?

Point RE

1. A and B of Example
2. A and B of Example, X and Y of Sub1
3.
4.

Referencing environments in dynamic-scoped languages

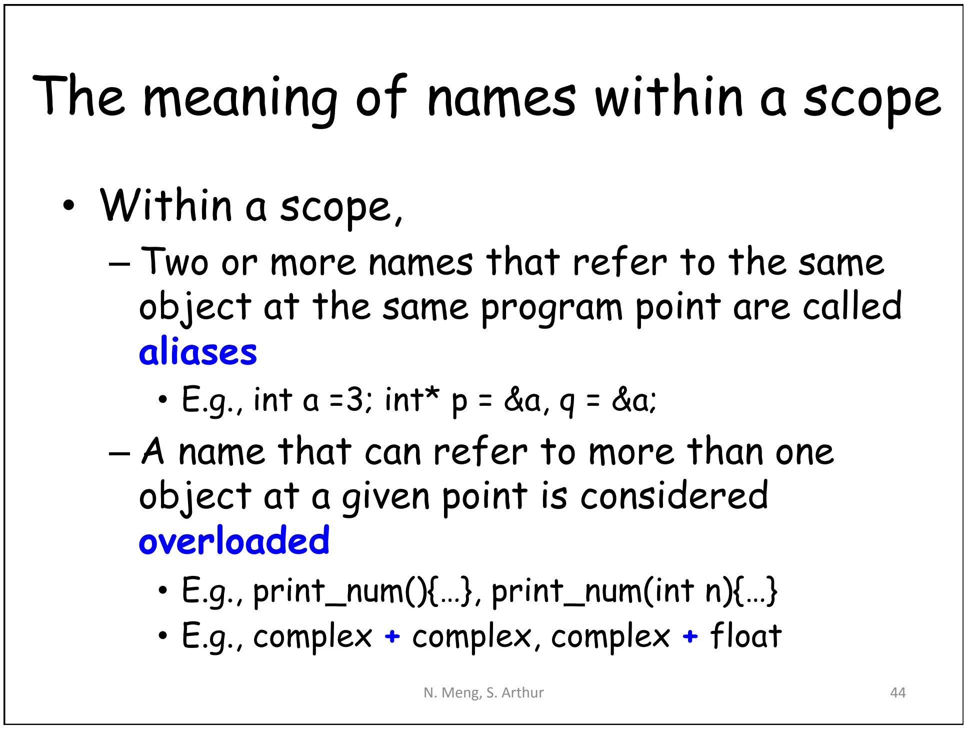
- A subprogram is **active** if its execution has begun but has not yet terminated
- The referencing environments of a statement in a dynamically scoped language is the locally declared variables, plus the variables of all other subprograms that are currently active
  - Some variables in active previous subprograms can be hidden by variables with the same names in recent ones



An Example

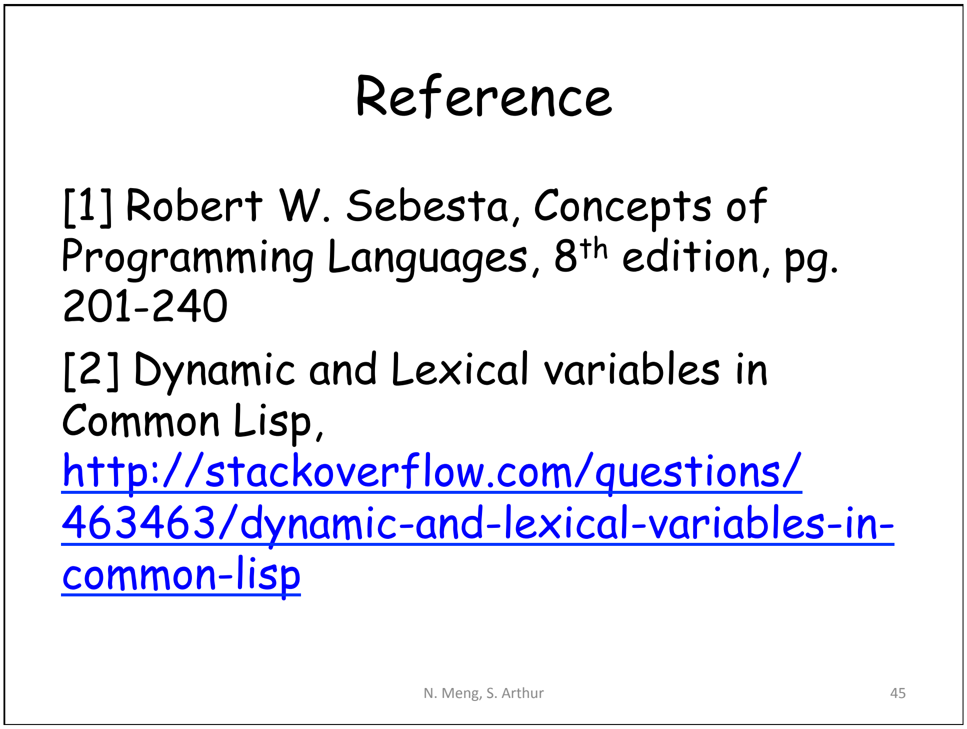
```
void sub1() {
   int a, b;
   …  ←-------------------------1
}  /* end of sub1 */
void sub2() {
  int b, c;
  …        ←------------------2
   sub1();
}   /* end of sub2 */
void main() {
   int c, d;
   …         ←---------------3
 sub2();
}   /* end of main */
```

What are the referencing environments of the indicated program points?



The meaning of names within a scope

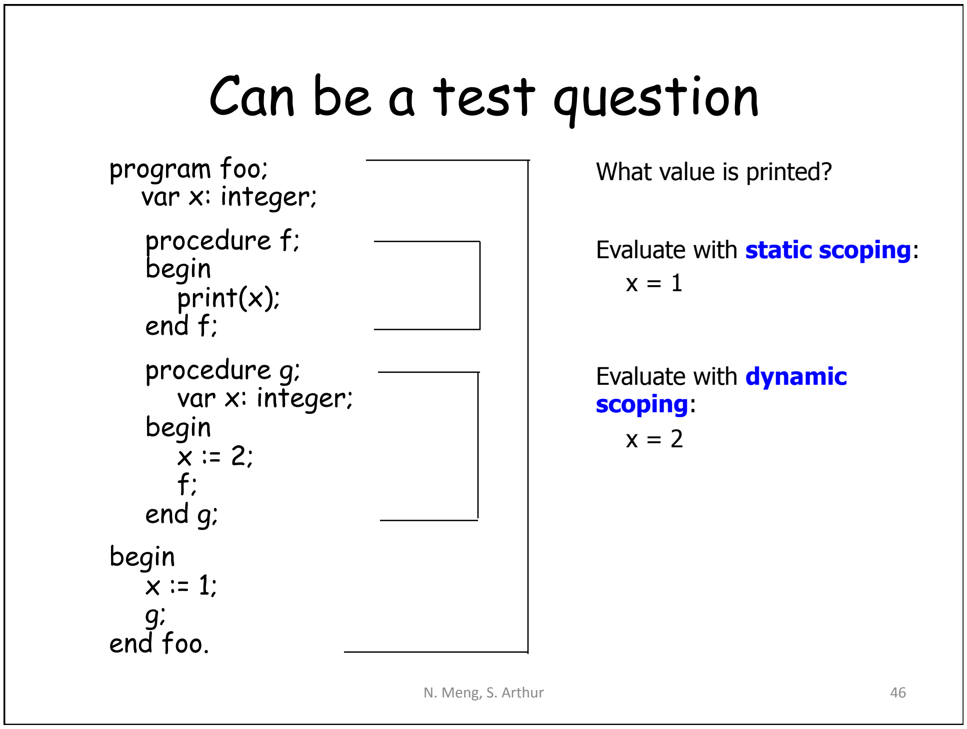
- Within a scope,
  - Two or more names that refer to the same object at the same program point are called **aliases**
    - E.g., int a =3; int* p = &a, q = &a;
  - A name that can refer to more than one object at a given point is considered **overloaded**
    - E.g., print_num(){…}, print_num(int n){…}
    - E.g., complex **+** complex, complex **+** float



Reference

[1] Robert W. Sebesta, Concepts of Programming Languages, 8th edition, pg. 201-240

[2] Dynamic and Lexical variables in Common Lisp, http://stackoverflow.com/questions/463463/dynamic-and-lexical-variables-in-common-lisp



Can be a test question

```
program foo;
   var x: integer;

   procedure f;
   begin
      print(x);
   end f;

   procedure g;
      var x: integer;
   begin
      x := 2;
      f;
   end g;

begin
   x := 1;
   g;
end foo.
```

What value is printed?

Evaluate with **static scoping**:
x = 1

Evaluate with **dynamic scoping**:
x = 2